# Optimisation des Algorithmes

**Objectif de la séance :** Éviter les algorithmes naïfs ou directs, au profit d'algorithmes plus élaborés pour lesquels l'analyse de complexité fournit de meilleurs résultats.

| | |
|---|---|
| 😎 | Application directe du cours |
| 😀 | Facile |
| 😖 | Moyen |
| 💀 | Difficile |

## 1 Retour sur les tris

### Exercice 1 : Tri rapide (quicksort) 😀

Dans le TD précédent, nous avons vu plusieurs algorithmes de tri sur les listes. On remarque que les tris intuitifs ont une complexité de l'ordre de $O(n^2)$, où $n$ est la taille de la liste d'entrée. D'autre part, les tris plus élaborés à base de "diviser pour régner", comme le tri fusion, sont plus intéressants car ils ont une complexité $O(n \log n)$. Nous commençons ce TD par un autre exemple de tri s'inspirant de la méthode "diviser pour régner" : le tri rapide, ou quicksort.

Le principe du tri rapide est de diviser la liste à l'aide d'un élement pivot.

1. Partager la liste selon la valeur v d'un élément arbitraire (le pivot), en pratique le premier élément de la liste, en deux sous-listes respectivement des éléments inférieurs ou égaux à v et des éléments de valeur supérieure à v.
2. Trier séparément les deux sous-listes extraites en suivant la même méthode.
3. Réarranger les sous-listes triées avec l'élément de référence (le pivot).

**Question 1.** Écrire une fonction `split` qui prend en argument une liste `l` et un entier `v`, et qui renvoie un couple `l1, l2` où `l1` contient les éléments de `l` inférieurs (ou égaux) à `v` et `l2` contient les éléments de `l` supérieurs (strictement) à `v`.

Par exemple, `split([2,3,8,1,5,4],4)` devra renvoyer `[2,3,1,4], [8,5]`.

**Question 2.** Écrire une fonction `quicksort` qui prend en argument une liste `l` et renvoie une copie triée de `l` suivant l'algorithme du tri rapide.

**Question 3.** On définit une opération élémentaire comme étant l'ajout d'un élément à une liste. Combien d'opérations élémentaires faut-il à la fonction `quicksort` pour trier une liste de taille $n$, en supposant qu'à chaque fois que l'on la divise en deux avec `split`, les deux listes obtenues `l1, l2` ont chacune pour taille $\approx n/2$ ?

*Indication.* On pourra utiliser le résultat suivant, qui est un cas particulier d'un énoncé plus général appelé "master theorem" :

*Si* $T : \mathbb{N} \to \mathbb{N}$ *est une fonction telle que* $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, *alors* $T(n) = O(n \log n)$.

**Question 4.** Que se passe-t-il si on utilise `quicksort` sur une liste déjà triée ?

## Exercice 2 : Recherche du $k$-ème élément 😀

**Question 1.** En s'inspirant de l'algorithme du tri rapide, implémenter une fonction `kth` qui prend en argument une liste `l` et un entier `k` (supérieur ou égal à 1) et qui renvoie le $k$-ème plus petit élément de la liste `l`.

Par exemple, `kth([1,3,2,1,4,2],4)` devra renvoyer `2`.

On pourra réutiliser la fonction `split` de l'exercice précédent, mais on s'interdira de commencer par trier la liste.

**Question 2.** Analyser la complexité de la fonction `kth` dans les deux cas étudiés dans l'exercice précédent.

# 2 Plus petit et plus grand élément d'une liste

## Exercice 3 : Méthode naïve 😎

**Question 1.** Écrire une fonction `minL` (respectivement `maxL`) qui prend en argument une liste d'entiers naturels non vide `L` et renvoie son élément minimal (respectivement maximal).

- ▷ `minL([0,3,1,7,0,8,12,4])` renvoie `0`.
- ▷ `maxL([0,3,1,7,0,8,12,4])` renvoie `12`.

On s'autorisera à utiliser les fonctions `min` et `max`, prédéfinies en Python, sur les entiers. Par exemple, `min(2,3)` renvoie `2`.

Quelle est la complexité de ces fonctions en nombre de comparaisons ?

**Question 2.** Écrire une fonction `min_maxL` qui prend en argument une liste d'entiers naturels non vide `L` et renvoie un couple constitué de l'élément minimal et l'élément maximal de `L`.

- ▷ `minL([0,3,1,7,0,8,12,4])` renvoie `(0,12)`.

Quelle est la complexité de cette fonction, en nombre de comparaisons ?

## Exercice 4 : Diviser pour régner 😀

On rappelle que l'approche "diviser pour régner" consiste à décomposer un problème en plusieurs sous-problèmes plus faciles, à résoudre récursivement ces sous-problèmes, puis à utiliser leurs solutions pour répondre au problème initial.

En suivant une approche "diviser pour régner" proposer une fonction `min_max_DR` qui prend en argument une liste non vide d'entiers naturels `L` et calculant le couple constitué de l'élément minimal et l'élément maximal.

*Indication* : on pourra utiliser les fonctions Python prédéfinies `min` et `max` qui prennent en arguments deux nombres et renvoient respectivement leur minimum et leur maximum. Pour les calculs de complexité, on considérera que ces fonctions ont un coût de 1 comparaison.

Quelle est la complexité (en nombre de comparaisons d'éléments) ?

# 3 Un grand classique

## Exercice 5 : La fonction $(x, n) \mapsto x^n$ 😖

**Question 1.** En utilisant l'approche la plus simple possible, implémenter la fonction `power` : $(x, n) \mapsto x^n$, où `x` est un flottant et `n` un entier positif. On s'interdira ici d'utiliser des fonctions prédéfinies comme `**`, `math.pow` ou `numpy.power`. Analyser la complexité en $n$ de votre algorithme. Il s'agit du nombre $T(n)$ de calculs nécessaires pour retourner `power(x,n)`, à `x` fixé.

**Question 2.** En utilisant une approche diviser pour régner, implémenter une autre fonction `power2` et analyser sa complexité, qui devra être logarithmique. Pourquoi est-ce que cette solution est bien meilleure que la première ?

*Indications.* Si on connaît $x^m$, comment calculer efficacement $x^{2m}$ ? Pour le calcul de complexité, on pourra commencer par montrer par récurrence sur $k$ que si $2^k \leq n < 2^{k+1}$, alors $T(n) \leq 2k + 2$.

# 4 Somme d'une sous-matrice

## Exercice 6 : Méthode directe 😀

L'objectif des prochains exercices est de réaliser un algorithme qui calcule la somme des éléments d'une sous-matrice d'une matrice de nombres.

Pour ces exercices, nous n'allons pas utiliser des libraires adaptées au calcul de matrices telles que `numpy`. Une matrice $M$ de $l$ lignes et de $c$ colonnes sera représentée en Python comme une liste de $l$ listes de taille $c$. Une sous-matrice de $M$ est définie par un intervalle de lignes de $l_1$ à $l_2$ et un intervalle de colonnes de $c_1$ à $c_2$, où $0 \leq l_1 \leq l_2 < l$ et $0 \leq c_1 \leq c_2 < c$. Notez qu'en numérotant les lignes de 0 à $l-1$ et les colonnes de 0 à $c-1$, nous avons adopté la notation Python. Par exemple, la matrice

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

sera représentée par la liste `[[1,2,3],[4,5,6]]`, ou plus visuellement :

```
[[1,  2,  3],
 [4,  5,  6]]
```

**Question 1.** Implémenter une fonction `ismatrix` censée prendre en argument une liste de listes d'entiers `m` et renvoyer un triplet `(b,l,c)` composé

- ▷ D'un booléen `b` qui vaut `True` uniquement si `m` représente une matrice.
- ▷ D'un entier `l` qui donne le nombre de lignes de `m` (dans le cas où `m` est bien une matrice).
- ▷ D'un entier `c` qui donne le nombre de colonnes de `m` (dans le cas où `m` est bien une matrice).

Par exemple :

- ▷ `ismatrix([[1,2,3],[4,5,6]])` renverra le triplet `(True, 2, 3)`.
- ▷ `ismatrix([[1,2,3],[4,5])` renverra le triplet `(False, None, None)` où `n` et `m` sont des nombres arbitraires de votre choix.

On commencera par donner l'algorithme en pseudo-code, puis on programmera la fonction `ismatrix` en Python en faisant les vérifications nécessaires quant au typage des arguments (revoir TD1 si besoin).

**Question 2.** Implémenter une function `sommation_dir` qui prend en argument une liste de liste d'entiers `m`, des entiers `l1`, `l2`, `c1`, `c2` et qui évalue la somme des éléments de la sous-matrice de `m` constituée des lignes `l1` à `l2` et des colonnes `c1` à `c2`.

▷ `sommation_dir([[1,2,3],[4,5,6]],1,1,0,1)` renvoie `9`.
▷ `sommation_dir([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]],1,3,1,2)` renvoie `6`.

Il est demandé de retourner un message d'erreur en cas de mal-formation des arguments. Quelle est la complexité de l'algorithme ?

## Exercice 7 : Version optimisée

**Question 1.** Implémenter une fonction `table_sommation` qui prend en argument une matrice `m` et renvoie une matrice `ts` de mêmes dimensions vérifiant

$$\texttt{ts[i][j]} = \sum_{0\leq \texttt{u}\leq \texttt{i}, 0\leq \texttt{v}\leq \texttt{j}} \texttt{m[u][v]}$$

En cas de mal-formation des arguments, la fonction `table_sommation` renvoie un message d'erreur. Il est attendu que la complexité de la fonction soit en $O(n \times m)$ où $n$ est le nombre de lignes et $m$ le nombre de colonnes.

▷ `table_sommation([[1,2,3],[4,5,6]])` renvoie `[[1,3,6],[5,12,21]]`.
▷ `table_sommation([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])` renvoie `[[1,2,3,4],[2,4,6,8],[3,6,9,12],[4,8,12,16]]`.

**Question 2.** En considérant que la table de sommation `ts` d'une matrice `m` est précalculée, implémenter une fonction `sommation_tab` qui prend en arguments `m`, `l1`, `l2`, `c1`, `c2` et qui renvoie la somme des éléments de la sous-matrice `[l1..l2] x [c1..c2]` de `m`. Cette fonction a les mêmes entrées/sorties que celle de l'exercice précédent, mais cette fois, il est attendu que la complexité soit en $O(1)$. Cette approche peut être avantageuse si l'on est amené à faire appel de nombreuses fois au calcul de la somme des éléments d'une sous-matrice. En effet, pour ce genre de problèmes, on va d'abord effectuer une étape d'initialisation où l'on calcule une bonne fois pour toutes la table de sommation, ce qui permet ensuite d'avoir des accès très rapides aux données.

# 5 Recherche de deux éléments d'une liste à partir de leur somme

On souhaite répondre à la question suivante : étant donné une liste `l` et un entier `x`, existe-t-il deux éléments de `l` d'indices distincts dont la somme est égale à `x` ?

## Exercice 8 : Méthode naïve

**Question 1.** Écrire une fonction `testSum` qui prend en argument une liste `l` et un entier `x` et renvoie un booléen qui indique si il existe deux éléments de `l` d'indices distincts dont la somme est égale à `x`. L'algorithme doit être le plus simple possible et se baser sur des comparaisons. La complexité dans le pire des cas doit être de $\Theta(n^2)$ comparaisons.

**Question 2.** Prouver que la complexité dans le pire des cas est en $\Theta(n^2)$.

## Exercice 9 : Solution optimisée

Il est possible de répondre à ce problème avec une complexité en $O(n \log n)$ comparaisons dans le pire des cas.

**Question 1.** Écrire une fonction `testSumOpti` qui renvoie le même résultat que `testSum`, mais avec une complexité en $O(n \log n)$ dans le pire des cas. On pourra utiliser le tri rapide de Python `sorted` qui prend en argument une liste et renvoie la liste triée dans l'ordre croissant en complexité $O(n \log n)$.

**Question 2.** Justifier proprement que la complexité dans le pire des cas est bien en $O(n \log n)$.

## 6 Sous-liste de somme maximale

On cherche à résoudre le problème suivant. Étant donné une liste d'entiers `l`, une sous-liste de `l` est une liste de la forme `l[i:j]` où $0 \leq$ `i` $<$ `j` $\leq$ `len(l)`. Quelle est une sous-liste de somme maximale de `l` ? Autrement dit, comment trouver deux `i` et `j` tels que $\sum_{\texttt{i} \leq k < \texttt{j}} \texttt{l}[k]$ est maximale ?

Par exemple, la liste $(1, -1, 1, 2, 3, -2)$ a pour sous-liste de somme maximale $(1, 2, 3)$ de somme 6. Ce n'est pas la seule possibilité, la sous-liste $(1, -1, 1, 2, 3)$ a aussi pour somme 6.

Ce problème peut être résolu par beaucoup d'algorithmes de complexité très variable. Nous allons en voir trois.

### Exercice 10 : Méthode brutale 😀

**Question 1.** Décrire informellement l'algorithme de type force brute (= tester toutes les combinaisons possibles) le plus simple qui vous vient à l'esprit pour répondre à la question posée. Estimer sa complexité.

**Question 2.** Écrire une fonction `maxsublist` qui implémente cet algorithme.

**Question 3.** Si à la question précédente vous avez trouvé une complexité de $\Theta(n^3)$, il est encore possible d'améliorer facilement votre algorithme. En économisant des opérations lors du calcul des sommes des sous-listes, modifier l'algorithme pour que sa complexité soit $\Theta(n^2)$.

### Exercice 11 : Diviser pour régner 💀

En fait, la complexité optimale pour résoudre ce problème est $\Theta(n)$. Ici, on va seulement montrer qu'il est possible de le résoudre en complexité $\Theta(n \log n)$ grâce à la méthode "diviser pour régner".

On cherche à écrire une fonction `optisublist` qui prend en argument une liste `l` et deux indices `low` et `high`, et qui renvoie la sous-liste de somme maximale *incluse dans* `l[low:high]`.

- On *divise* en séparant la liste `l[low:high]` en deux listes de même taille (on coupe au milieu `mid` $= \frac{\texttt{low+high-1}}{2}$).
- On *règne* en trouvant récursivement la sous-liste de somme maximale dans la première liste, `optisublist(l,low,mid+1)` puis dans la deuxième, `optisublist(l,mid+1,high)`.
- Enfin, on utilise ces deux sous-résultats pour déduire quelle est la sous-liste de somme maximale dans `l[low:high]`. C'est l'étape difficile.

On définit ensuite tout simplement la fonction `maxsublist3(l) = optisublist(l,0,len(l))`.

**Question 1.** Implémenter la fonction `optisublist`.

**Question 2.** Prouver que la complexité de `maxsublist3` est en $\Theta(n \log n)$, où $n$ est la taille de la liste d'entrée.

# 7 Diviser pour reigner (avancé)

"Diviser pour régner" est une technique algorithmique qui consiste à décomposer un problème en plusieurs sous-problèmes plus faciles (diviser), à résoudre récursivement ces sous-problèmes (réfgner), puis à utiliser leurs solutions pour répondre au problème initial (combiner). Il s'agit ici de problèmes légèrement plus difficiles.

## Exercice 12 : Calcul d'une ligne de gratte-ciels (skyline)

Le problème de la ligne de gratte-ciels (skyline problem) consiste à partir d'un ensemble de gratte-ciels (donnés sous la forme d'un triplet $[x, h, y]$ avec $x$ abscisse gauche, $h$ hauteur, et $y$ abscisse droite) à construire la ligne de séparation entre les bâtiments et le ciel sous la forme $[x1, h1, x2, h2, ..., xi, hi, ..., xn, 0]$ représentant la ligne de séparation sous forme d'escalier.

Pour l'exemple de la figure 7, la ligne de gratte-ciels associée à l'ensemble de gratte-ciels

$$[[3, 13, 9], [1, 11, 5], [12, 7, 16], [14, 3, 25], [19, 18, 22], [2, 6, 7], [23, 13, 29], [23, 4, 28]]$$

est :

$$[1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0]$$

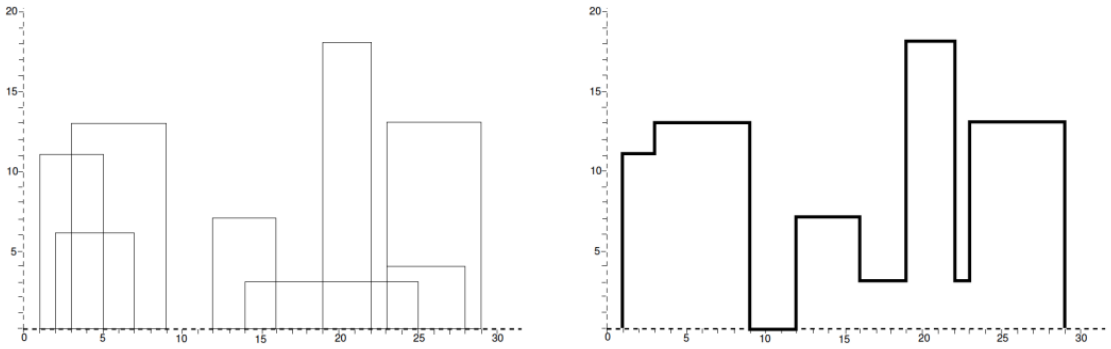


Figure 1: Ligne de gratte-ciels.

**Question 1 :** Ecrire l'algorithme qui à partir d'une liste de gratte-ciels, fournit la ligne de gratte ciel.

**Question 2 :** Quelle en est la complexité ? Peut-on faire mieux ?

## Exercice 13 : Points 2D les plus proches

Considérons un ensemble $E$ de $n$ points $p$ donnés par leurs coordonnées planaires $(p_x, p_y)$ vérifiant (pour des soucis de simplicité) que deux points distincts $p^1$ et $p^2$ de $E$ ont des abscisses et des ordonnées différentes ($p^1_x \neq p^2_x$ et $p^1_y \neq p^2_y$).

**Question 1 :** Ecrire une fonction naïve **dmin($L$):list** $\rightarrow$ **float** qui calcule la plus petite distance possible entre deux points distincts de $E$. L'entrée est une liste de liste de la forme $[[x1, y1], [x2, y2], \ldots, [xi, yi], \ldots, [xn, yn]]$.

On donne le code de calcul de la distance entre deux points :

```python
import math

def distance(p1,p2):
    return math.sqrt(abs(p1[0] - p2[0])**2 + abs(p1[1] - p2[1])**2)
```

On propose aussi un code pour générer une liste de points pseudo-aléatoires:

```python
import random
random.seed(0)

def build_point(min_x, max_x, min_y, max_y):
    """
    input :
        min_x abscisse minimale
        max_y abscisse maximale
        min_y ordonnee minimale
        max_y ordonnee maximale
    output :
        un point pseudo-aleatoire
        d'abscisse comprise entre min_x et max_y
        et d'ordonnee comprise entre min_y et max_y
    """
    x = min_x + random.random() * (max_x - min_x)
    y = min_y + random.random() * (max_y - min_y)
    return [x,y]

def build_points(min_x, max_x, min_y, max_y, nb_points):
    """
    input :
        min_x abscisse minimale
        max_y abscisse maximale
        min_y ordonnee minimale
        max_y ordonnee maximale
        nb_points nombre de points souhaites
    output :
        une liste de points pseudo-aleatoires
        d'abscisse comprise entre min_x et max_y
        et d'ordonnee comprise entre min_y et max_y
    """
    return [build_point(min_x, max_x, min_y, max_y) for _ in range(nb_points)]

points = build_points(-50,50,-50,50,100)
```

**Question 2 :** Quelle est la complexité en temps de votre algorithme en supposant que le calcul de distance entre deux points se fait en temps constant ?

**Question 3 :** Proposer un algorithme de type *diviser pour régner* pour résoudre le problème de recherche des plus proches points en 2D.

**Question 4 :** Montrer que la complexité de l'algorithme proposé pour la question précédente est en $O(n \log n)$. Il est attendu que l'étape de combinaison des solutions des sous-problèmes soit soigneusement décrite et justifiée.

**Question 5:** Implémenter la version "diviser pour régner" de l'algorithme en Python.